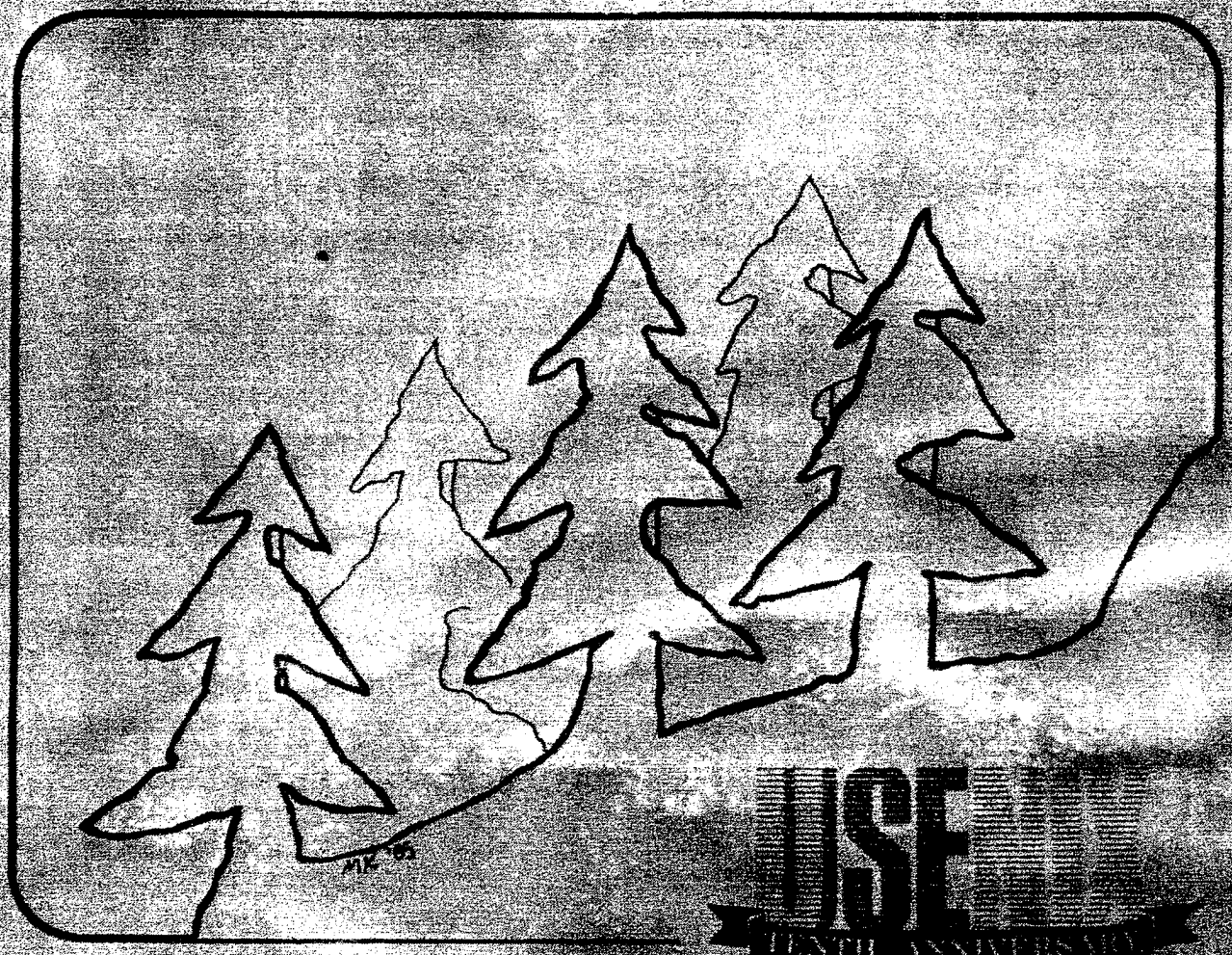

Summer
1985

Usenix Conference & Exhibition

Portland, Oregon

Conference Proceedings

For additional copies of these proceedings, write:
USENIX Association
P.O. Box 7
El Cerrito, CA 94530 USA
Price: $25.00 plus $25.00 for overseas mail

Implementing Loosely Coupled Functions on Tightly Coupled Engines.

Jack Inman

Sequent Computer Systems
14360 NW Science Park Drive
Portland, Oregon 97229

ABSTRACT

This paper describes the experiences of porting UNIX[1] 4.2bsd networking functions to DYNIX,[2] a tightly coupled, multiprocessor implementation of UNIX 4.2bsd. The paper identifies networking issues in a multiprocessor system and generalizes multiprocessor system concepts by way of relating examples of them encountered in the Interprocess Communications (IPC) portions of the concurrently executing DYNIX kernel. It first briefly describes the UNIX 4.2bsd IPC model, then the DYNIX multiprocessor model and how it is applied to IPC. It further discusses some of the things that can go wrong, how these situations were discovered and corrected, and finally gives examples of the general issues and effective implementations.

1. Introduction

The author is responsible for the IPC portions of Sequent's DYNIX kernel. DYNIX is a kernel based on UNIX 4.2bsd that executes concurrently on multiple processors. The development of DYNIX IPC involved "porting" the UNIX 4.2 IPC architecture and algorithms to a multiprocessor system. The nature of the development, i.e. it being a port of existing architecture and algorithms, offers the benefits of many algorithmic solutions already implemented in UNIX 4.2bsd. Furthermore, porting suggests the ability to quickly absorb other protocol implementations under UNIX 4.2bsd as they are developed. A faithful port also offers a good degree of assurance that the semantics of the protocols from the points of view of both the user and the network are maintained. The drawbacks to the approach however are that 1) concurrent programming is not an easy thing to do anyway, 2) the base implementation executes in a monoprocessor system and thereby dictates certain semantics, and 3) characteristics of networking systems introduce additional multiprocessor pitfalls. It requires continuous judgement on when to port an algorithm, a set of semantics, or a particular data structure, and when to redesign or enhance the design to accommodate concurrent processing. DYNIX is evidence of the success of this effort and this paper shares some of the lessons learned.

---

1 UNIX is a trademark of AT&T Bell Laboratories.

2 DYNIX is a trademark of Sequent Computer Systems, Inc.

2. Rationale for the DYNIX IPC Model

A good question to answer at this point is why do this? Since multiple processors are available in the DYNIX host, and UNIX 4.2bsd provides useful protocol implementations implemented within the kernel, it is advantageous to implement IPC services within the DYNIX kernel to execute in a multiprocessor host. Furthermore, with a general multiprocessor model, greater advantage can be taken of concurrency as the number of processors increases. This means for example that as networking needs grow, more processors can be effectively added.

There are alternative approaches. For example, a loosely coupled approach, in which the IPC portions of the kernel are moved to intelligent controllers is one alternative. This approach is popular in monoprocessor environments because it off loads the single host processor of a lot of processing requirements. However, in general it does not cover all interesting cases since significant IPC services (notably pipes) must be executed in the hosts. Furthermore, such loosely coupled implementations have disadvantages in internetworking configurations.

A different IPC model could have been chosen such as "transparent" distributed file system, but this suffers from nonstandardization. Such distributed file systems typically work well only with themselves. Interoperability and compatibility with existing equipment are key objectives of DYNIX.

There are benefits of implementing the IPC services within DYNIX. The major benefit of course is performance. This is not to say that a stream of bytes can be transferred from point A to point B quicker using DYNIX than a monoprocessor implementation of the same protocols. Consider, the sequential algorithms are essentially equivalent, being ported from a monoprocessor implementation. Furthermore the class of processor (National Semiconductor's Series 32000) is essentially equivalent to a VAX 750[3] in silicon. It is difficult to imagine performing any better than a comparable monoprocessor implementation with respect to how long a particular network transaction takes end to end. The issues are concurrency and total system performance. I.e. how much total work gets completed by the system including, or especially, network related work.

Significantly, concurrency allows user processes to make progress while network services are being executed. For example, user response time is less affected by network activity since a separate processor can do the network chores. This generally results in the system being able to make more IPC service requests, thereby making more work available more often. Concurrency further allows different pieces of the network services to be executed by multiple processors concurrently, thereby improving total throughput due to better service to the network.

2.1 Concurrency and "Layered" Communications Model

Contemporary communications subsystems are "layered". They are often modeled after the ISO Open Systems Interconnect Architecture [Zimmerman 1980]. There are many benefits to this layering and examples of its effectiveness.

It is natural to imagine the application of multiple, concurrently executing processors *vertically* through this model. For example, a processor might be allocated per protocol layer, or as often the case in practice, several layers might be grouped together in "macro" layers. Vertical concurrency is exemplified by several multiprocessor architectures, including loosely coupled architectures such as various intelligent front end network controller models. Although a degree of concurrency is achieved with these models, it is available only for network services and only applicable vertically through the ISO layers. In affect, the processor on the controller is not available for anything else. With the cost of microprocessors decaying so rapidly, this is acceptable idle time in some situations, but having the ability to perform the network function within a general purpose computer environment has many advantages.

3 VAX is a trademark of Digital Equipment Corporation

For example consider the effect of three processors allocated in a well balanced fashion to perform a network function such as a file transfer. One processor is busy performing user process tasks such as obtaining the data and communicating with the user. Another processor is busied managing network tasks such as packetizing the data, checksumming, and executing various protocol management algorithms. It is doing this for several streams of network data flow. The third processor is handling the low level interface to the transmission medium. This scenario creates an effective pipeline of processors doing a lot of work without having to context switch. The movement of any one piece of information is somewhat more expensive than it would be with only one processor executing the tasks, but the total amount of data that is transferred is more. In practice, of course, the situations are much more dynamic. In the long run, fair, dynamic allocation of processors to concurrently executing tasks improves total system throughput. DYNIX IPC accomplishes such concurrency with many more degrees of freedom than this simple example. The actual hardware on which DYNIX executes, the Balance 8000,[4] supports up to twelve NSC Series 32000 microprocessors. Several network functions execute concurrently as well. The implementation accommodates 1 to 12 processors and benefits from each incremental number of processors.

Using a tightly coupled multiprocessor model a dedicated controller processor is not required for network processing. This lowers the cost of the connection. For example, in the case of a Balance 8000, an interface to the Ethernet[5] does not require a controller per se. The Ethernet data link is simply one function of a multi-purpose controller board [Sequent 1984].

Horizontal concurrency is also possible. An obvious consideration in a multi-user and multiprocess environment such as UNIX is concurrent execution of user application protocols. Multiple communications functions such as file transfer and virtual terminal can execute on several processors sharing the same operating system services. Gateway servers might take advantage of multiple concurrently executing network interfaces. It is also possible for multiple concurrently executing protocol suites, such as XNS[6] and TCP/IP to operate over a single Ethernet data link. Transmit and receive functions can also execute concurrently, generally (but not always) independent of each other. Other concurrent execution possible includes timer events and network management functions such as routing table updates.

2.2 UNIX 4.2bsd Compatibility as a Goal

A significant objective of this port is to be fully compatible with UNIX 4.2bsd both from the user interface and perhaps more importantly, from the network interface. A user program that makes use of the UNIX 4.2bsd IPC interfaces should be readily portable to a Balance 8000 system. Existing (and future) applications of IPC services should also work on current (and future) DYNIX systems. Examples of such user applications are the "r commands" which UNIX 4.2bsd supports, viz. *rlogin, rsh* and *rcp*. From the Unix Programmer's Manual pages [UNIX 1983]:

rlogin (1C)	- remote login
rsh (1C)	- remote shell
rcp (1C)	- remote file copy

These applications are supported in DYNIX with virtually no changes other than recompilation. Furthermore, since compatibility is also achieved at the network interface, these applications function well across a network to other UNIX 4.2bsd implementations. That is to say, a Balance 8000 hosting DYNIX easily communicates with a VAX hosting UNIX 4.2bsd and vice versa. One can *rlogin, rsh,* or *rcp* files to and fro.

---

4 Balance is a trademark of Sequent Computer Systems, Inc.

5 Ethernet is a trademark of Xerox Corporation.

6 XNS is Xerox's Networking Systems protocol suite

For example, the Sequent company environment contains several different kinds of computer systems including multiprocessor DYNIX systems and monoprocessor UNIX 4.2bsd systems. Using these applications, all that is needed is a nickname for the machine. Since the interfaces are the same, it is easy to use them. Once logged in, it is difficult to tell a monoprocessor UNIX 4.2bsd machine from a multiprocessor DYNIX machine (except for the response time and load average statistics). In the Sequent company environment most people in the company have reason to use resources on several different machines. This includes people that do not have development responsibilities. Interesting evidence to support the conclusion that most users access multiple hosts lies in the fact that one of the most used commands on the production machines is *hostname*[7] which is often placed in user's prompts to remind them where they are. The compatibility is sufficient to allow one to forget what type of machine it is that they are talking to (until they try to execute a NSC Series 32000 binary file on a VAX).

2.3 Network Interoperability

A major rationale for porting the supported Internet protocols is the degree of interoperability which standards can achieve. Balance 8000 can immediately participate on networks of computers executing UNIX 4.2bsd, but also on many networks with computers that support implementations of the Internet protocols over the Ethernet. These include the DARPA[8] functions *telnet* (comparable to *rlogin*) and *ftp* (comparable to *rcp*). From the UNIX Programmer's Manual pages [UNIX 1983]:

ftp (1C) - file transfer program
tftp (8C) - DARPA Trivial File Transfer Protocol
telnet (1C) - user interface to the TELNET protocol

Using these applications, DYNIX can also communicate through existing gateways to other implementations. It is a significant win for a new product to be able to "talk" to existing solutions without having to develop special hardware or software. Balance 8000 and DYNIX achieve this goal.

3. The UNIX 4.2bsd IPC Model

The following briefly describes details of the UNIX 4.2bsd Communications Architecture. In this architecture IPC services are integrated into the kernel. Consequently IPC services are also integrated into the concurrent DYNIX kernel. It is assumed that the reader is somewhat familiar with the UNIX 4.2bsd model such that brief context is sufficient and is referred to the references for further insight [Leffler 1983, Leffler 1983B]. Also, since DYNIX implements the same general architecture with the enhancement of concurrent execution, the terms UNIX 4.2bsd, DYNIX, and kernel are used *somewhat interchangeably*.

With respect to IPC the kernel provides functions that:

[1] Define and manage data structures to support the communications architecture.

[2] Initialize the communications system.

[3] Provide the system interfaces that map user requests to communications services.

---

7 hostname (1) - set or print name of current host system [UNIX 83].

8 DARPA is the Defense Advanced Research Projects Agency network which defines a set of standards for computer to computer communications.

[4] Provide the system interfaces that map communications services to user requests.

[5] Manage the kernel's communications system buffers *(mbuf's)*.

[6] Manage the movement of data from kernel space, to user space.

[7] Provide timer events.

At the base of the communications definition is the notion of communications **domain**. [Leffler 1983B] defines a communication domain as "an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets."

In the kernel the domain structure consists of a linked list of the domain entries supported by the system. Consequently, multiple domains are supported by the architecture. The domain entries are used to find the appropriate **protocol switch**, *struct protosw*. Each domain includes an array, *protosw*[], of *protosw* entries. A single domain supports multiple protocols each one represented by a *protosw* entry.

An **Address Family** specifies a communications domain. For example, the Internet domain is identified by the value AF_INET. There are two communications domains currently supported in DYNIX:

[1] UNIX domain (AF_UNIX),

[2] Internet domain (AF_INET).

The AF_UNIX domain is used for UNIX to UNIX local interprocess communication. It for example includes communication via UNIX pipes. The AF_INET domain is used for both local and remote interprocess communication using the DARPA protocols.

There are three types of communications available represented by **socket types**. These are *stream* (SOCK_STREAM), *datagram* (SOCK_DGRAM), and *raw* (SOCK_RAW) socket types.

AF_UNIX supports three *protosw* entries, one for a SOCK_STREAM, one for a SOCK_DGRAM, and one for a SOCK_RAW socket interface. Pipes are included in the SOCK_STREAM definition. AF_INET has five *protosw* entries corresponding to the Internet Protocol (IP), Internet Control Message Protocol (ICMP), User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and a "raw" internet protocol (rip_). Not all of the information is required for every *protosw* entry. Entries are NULL if not needed.

At system initialization, *domaininit()* is executed to link the domain data structures and call the protocol initialization procedures via the *protosw*[] entries. In this way the domain structure is used to define a protocol family's features.

For purposes of further discussion the IPC services are divided into Protocol Engines, Socket Management, and Network Interface Management.

3.1 Protocol Engines

This paper uses the notion of **protocol engines** to represent the protocol specific functions executed by the kernel. This includes the algorithms and data structures used to implement a particular set of protocols. Protocol engines are defined in part by the routines specified in a domain's *protosw*[]. They perform the functions of the protocol including transmission and delivery of data, and control of that transmission and delivery. A *protosw* entry is defined [Leffler 1983]:

```
struct protosw {
  short                         pr_type;          /* socket type */
  short                         pr_family;        /* protocol family */
  short                         pr_protocol;      /* protocol number */
  short                         pr_flags;         /* see below */
    /* protocol engine handles */
  int                           (*pr_input)();    /* input (from below) */
  int                           (*pr_output)();   /* output (from above) */
  int                           (*pr_ctlinput)(); /* control input (from below) */
  int                           (*pr_ctloutput)(); /* control output (from above) */
    /* user-protocol hook */
  int                           (*pr_usrreq)();   /* user request: list below */
    /* utility hooks */
  int                           (*pr_init)();     /* initialization hook */
  int                           (*pr_fasttimo)(); /* fast timeout (200ms) */
  int                           (*pr_slowtimo)(); /* slow timeout (500ms) */
  int                           (*pr_drain)();    /* flush excess space */
};
```

Socket management and protocol engines routines pass commands to the protocol engine via the protocol engine handles. The commands passed to the protocol engines include:

```
#define PRU_ATTACH          /* attach protocol to up */
#define PRU_DETACH          /* detach protocol from up */
#define PRU_BIND            /* bind socket to address */
#define PRU_LISTEN          /* listen for connection */
#define PRU_CONNECT         /* establish peer connection */
#define PRU_ACCEPT          /* accept peer connection */
#define PRU_DISCONNECT      /* disconnect from peer */
#define PRU_SHUTDOWN        /* won't send any more data */
#define PRU_RCVD            /* data taken; more room now */
#define PRU_SEND            /* send this data */
#define PRU_ABORT           /* abort */
#define PRU_CONTROL         /* control ops on protocol */
#define PRU_SENSE           /* return status into m */
#define PRU_RCVOOB          /* retrieve out of band data */
#define PRU_SENDOOB         /* send out of band data */
#define PRU_SOCKADDR        /* fetch socket's address */
#define PRU_PEERADDR        /* fetch peer's address */
#define PRU_CONNECT2        /* connect two sockets */
#define PRU_FASTTIMO        /* 200ms timeout */
#define PRU_SLOWTIMO        /* 500ms timeout */
#define PRU_PROTORCV        /* receive from below */
#define PRU_PROTOSEND       /* send to below */
```

These routines are the kernel's handles into a protocol engine. Socket management uses the appropriate *pr_usrreq()* handle to service user process requests (requests from "above") and network processes use them to access protocol engines from "below". They are also used from within to execute initialization and timer events.

A protocol engine is exemplified by the Internet protocol engine for the Internet protocol set. It supports the DARPA standard Transmission Control Protocol (TCP), User Datagram Protocol

(UDP), Internet Control Message Protocol (ICMP), and the Internet Protocol (IP).

Note, that the Communications Architecture is independent of actual protocols employed, and it accommodates multiple communications protocols within the same system. Although there is a defined set of protocol engines currently supported by DYNIX, the architecture supports other protocol engines. There is further analysis that proposes some minor improvements to make the architecture even more general. [O'Toole 1985].

3.2 Socket Management

A key abstraction in the communications architecture is that of a **socket**. A socket is what a user process uses to access IPC services, such as send and receive. Socket management provides the binding of user processes to sockets. It therefore manages movement of data in and out of user processes and the kernel, creation and deletion of IPC resources, and reference to protocol engine services (i.e. the execution of the appropriate communications protocols).

A socket data structure contains pointers to the protocol control information, input and output data queues, socket state variables, and other variables used by the system. It is used by the kernel to process user requests, network requests, and time out requests. It is created as a result of a *socket()* UNIX 4.2bsd system call.

A program references a socket in order to affect interprocess communication. A socket is analogous to a file descriptor. In fact, internally sockets are defined as a type of file descriptor. Consequently, kernel file system services are used by the IPC implementation to help manage socket data structures.

For example, when a *socket()* system call is made, a file descriptor is acquired from the user process' available set of file descriptor resources. These are managed by the kernel just like any other file descriptor. When the socket abstraction is closed, either explicitly, or implicitly due to process termination, the kernel uses the file descriptor to initiate deallocation of IPC resources.

The IPC system calls available to a user process are described in the UNIX Programmer's Manual pages [UNIX 1983]:

read, readv (2)	- read input (from SOCK_STREAM)
write, writev (2)	- write on a file (to SOCK_STREAM)
close (2)	- delete a descriptor (E.g a socket)
pipe (2)	- create an IPC channel
ioctl (2)	- control device (or net parameter)
gethostname, sethostname (2)	- get/set name of current host
select (2)	- synchronous i/o multiplexing
socket (2)	- create an endpoint for IPC
connect (2)	- initiate a connection on a socket
accept (2)	- accept a connection on a socket
send, sendto, sendmsg (2)	- send a message from a socket
recv, recvfrom, recvmsg (2)	- receive a message from a socket
bind (2)	- bind a name to a socket
getsockopt, setsockopt (2)	- get and set options on sockets
listen (2)	- listen for connections on a socket
shutdown (2)	- shut down part of a FDX connection
socketpair (2)	- create a pair of connected sockets
getpeername (2)	- get name of connected peer
gethostid, sethostid (2)	- get/set unique id of current host
getsockname (2)	- get socket name

3.3 Network Interface Management

Protocol engines and socket management provide data to and receive data from any of several physical networks via network interfaces. Network interfaces define drivers provided for transmitting and receiving data over a network's physical media. The network interfaces control the hardware which does data transfer. Data passes through the network interfaces via input and output queues defined in network interface structures, *struct ifqueue* in *struct ifnet*.

Network interface output queues are typically serviced by transmit complete interrupt handling routines. Protocol engine routines queue output requests to the interface to be transmitted asynchronously. Received data is placed into input queues by receiver interrupt routines. These queues allow the protocol engines to execute at lower priority than hardware interrupt handling. On input, appropriate data link **clients** are started via **software interrupts** described below.

The most notable example of a hardware network interface is the Ethernet. Multiple Ethernet interfaces are supported in both UNIX 4.2bsd and DYNIX. Packets are received from an Ethernet interface and demultiplexed to clients by using the type field defined in the Ethernet standard [DEC 1982]. The most interesting data link client for discussion in this paper is the Internet protocol engine which receives Internet packets from one or several network interfaces. It processes the received data according to the appropriate protocol, E.g. TCP, UDP, or ICMP. It may also choose to perform internetwork routing and forward the data packet to another network interface.

UNIX 4.2bsd uses a mechanism referred to as a **software interrupt** to partition network functions. This allows lower level, but higher priority functions such as those done by the driver to execute quickly and schedule higher level functions such as the protocol engine functions for later execution at a lower priority. The data is queued into a network interface queue and when the situation permits, the higher level protocol engine executes. This facility provides good system response to the network interface when the network is busy.

4. The DYNIX Multiprocessor Model

DYNIX is a concurrently executing version of the UNIX 4.2bsd kernel. It provides three mutual exclusion (**mutex**) mechanisms, gates, locks and semaphores [Beck 1984]. The interfaces to these mechanisms include familiar "P and V" semantics from Dijkstra's *Proberen*and *Verhogen* ideas [Dijkstra 1965]. *p__xxx* attempts to acquire the resource, and *v__xxx* releases it.

Gates are used for low level, high priority processes such as memory management, low level structure management, and interface management. Critical sections protected by gates are kept to a minimum due to the fact that processors do not make progress while waiting for a gate to become available. Also there are a limited number of gates in the system supported by a proprietary silicon device, the System Link and Interrupt Controller (SLIC) [Fielland 1984]. A *p__gate()* results in a SLIC operation to acquire a gate. A *v__gate()* results in a SLIC operation to release it.

Locks are implemented using gates and are software managed mutual exclusion structures. *p__lock()* acquires the lock and *v__lock()* releases it. A processor can "spin" attempting to acquire a lock, but can be interrupted by a higher priority process if required. A processor execution priority level (SPL) is specified on the lock calls. However, keep in mind that SPL is insufficient for mutual exclusion in a multiprocessor system since other processors can be executing at the same, or even lower SPL concurrently [Bach 1984]. This is discussed further below.

The third mutual exclusion mechanism is the counting semaphore. A *p__sema()* acquires the semaphore if available and places the process on a queue if it is not available. A *v__sema()* starts the next process in line waiting for the semaphore. There is also a *vall__sema()* which starts all processes currently queued on the semaphore.

In addition to the standard "P & V" interfaces, DYNIX also provides "conditional" requests for locks and semaphores similar to those described in [Bach 1984], *cp_lock()* and *cp_sema()*. These interfaces are useful for cases where something else can be done rather than contend for the resource.

Another useful mutex interface is the *p_sema_v_lock()* call which allows a semaphore to be requested and a lock to be released in one call. This for example is useful to prevent semaphore structures from being deleted while in the process of being requested. It also helps to avoid "sleeping" while a lock is held.

4.1 DYNIX IPC Kernel Concurrency Model

The IPC portions of the kernel effectively make use of the DYNIX mutual exclusion mechanisms described to accommodate concurrency. The following describes their general use.

Semaphores are used to mutex user processes from each other, to signal when particular events occur such as connect or disconnect complete, and to contend for memory resources. A global semaphore is used to simplify some of the UNIX domain non-pipe IPC. Use of semaphores allows processes to wait for resources to become available, thereby allowing a processor to do something else until the resource is available. There are three semaphores defined for each socket. They are used for connection/disconnection signaling, and synchronization of requests for the input and output data queues.

Locks are used for lower level mutex generally involved with structures shared with protocol engines. For example, each socket structure has a lock associated with it. By assigning a lock per resource, unrelated user processes do not contend for the same resource for unrelated IPC. Locks are also defined for several protocol engine lists and the network interface queues.

Gates are used for the lowest levels of IPC services such as network buffer memory management, and low level network interface management. Gates are also used to mutex the software lock structures. IPC uses a total of three gates.

5. Interesting Networking Issues

Porting the IPC kernel to a multiprocessor system encounters many issues common to the general problems of implementing multiprocessor systems, but some issues are particularly germane for networking environments. Examples include network interrupts, unsolicited data and remote initiation of shared resource modification.

5.1 Network Interrupts

As described above, UNIX 4.2bsd implements software interrupts in order to provide good response to a busy network. DYNIX easily accommodates this feature and indeed, high level protocol engines execute concurrently with multiple network interface interrupt handlers. For input to a protocol client, data is placed into an input queue and a software interrupt is signaled. In a monoprocessor system the software interrupt is executed later at a lower priority. In a multiprocessor system it can execute immediately, or already be executing when the data arrives. Thus the first order of concurrent processing is easily achieved in DYNIX by properly managing the interface queues and software interrupt mechanism. This is relatively straight forward and uses a simple *p_lock()/v_lock()* protocol to enqueue and dequeue packets on the network interface.

This means that network interfaces can provide good service to network devices while protocol engine functions execute concurrently. The protocol engines avoid multiple instantiations of the software interrupt queue handlers, therefore, if the traffic warrants it, the processor continues to operate as a protocol engine. In a sense, the processor becomes a temporarily dedicated controller until something else more important requires the processor resource.

5.2 Unsolicited Data

An interesting characteristic of network subsystems is the frequent receipt of **unsolicited data.** This is described as "spontaneous input" in [Requa 1985] and is familiar to most developers of network or real time code. The issue is more pronounced in systems distributed on a Local Area Network (LAN) because typically packets of data arrive at random, usually bursty, intervals from multiple sources and potentially destined for multiple protocol engine clients. The receiving system must be ready to receive the packets or risk loosing them. Packet loss is not necessarily critical in networking subsystems since their purpose is to provide a level of deliverable service which assumes unreliable transfer. Nonetheless, it affects total system performance and missing packets is undesirable.

Typically, systems manage unsolicited data by buffering ahead. This can be as simple as two level buffering, where the network software immediately sets up the next input buffer upon receiving a block of network data. The more buffers available, the more back to back packets can be received without loosing them due to lack of buffers. However, it is impossible to anticipate how much data to expect for each packet since network data on a LAN consists of both large and small packets. Shoch's analysis shows Ethernet traffic for a comparable environment is bimodal [Schoch 1979]. This is true with TCP/IP also. In fact, the bimodalness of typical TCP/IP LAN traffic is so dramatic that "histograms" are formed simply by printing the decimal counts of the packet sizes seen. For example, the following is a picture of such analysis performed by monitoring Sequent's busy Ethernet under typical load:

```
total_packets -        27347608
packet counts per bucket ->
                    buck[0] = 15450762
                    buck[1] = 923132
                    buck[2] = 442860
                    buck[3] = 367880
                    buck[4] = 1052369
                    buck[5] = 9110664
```

The value of buck[0] represents the number of packets that contain under 64 bytes and buck[5] represents the number of packets that contain more than 1024 bytes. The decimal numbers themselves form a bimodal histogram. The buck[0], or short packet number represents 56% of the total packets and buck[5] or long packet number represents 33% of the total packets in this particular environment. The counts for [1] 64-128, [2] 128-256, [3] 256-512 are 2 decimal digits shorter than the counts for short packets and 1 decimal digit shorter than the counts for long packets. This is understandable considering protocol control packets and virtual terminal traffic are generally contained in short packets and file transfer generally attempts to use predominantly large packets.

This is of course just a snapshot of a particular period of network usage on a particular network and it changes with time and usage. For example, dumps across the network using the Unix 4.2bsd commands [UNIX 1983]:

```
rrestore (8C)    - restore file system dump across the network
rdump (8C)       - file system dump across the network
```

skew this more towards larger buffers.

The point is that for such unsolicited data, the system needs to be prepared for just about anything, at any time. Buffering ahead provides this preparation. DYNIX buffers ahead using chained 128-byte buffers. The number of these receive buffers is a configurable parameter that defaults to 200. The Ethernet driver attempts to keep this number of network buffers available for unsolicited input at all times. The more memory allocated to this function, the more packets can be queued during a burst of data. At the expense of tying up some system RAM, optimal

performance is ensured.

The replacement of network buffers does not always succeed and the network interface must be prepared to catch up at the next opportunity. This is accomplished by the network receiver interrupt handler requesting enough buffers to fill its pool. If a request for buffers is denied, it is picked up on the next receiver interrupt. The driver always keeps at least one buffer queued in order to keep things active, even if it has to discard the last packet received.

5.3 Remote Initiation of Resource Modification

A fundamental difference between tightly coupled systems and loosely couple systems is the degree of asynchronism. Loosely coupled systems are characterized by comparatively long delays between requests and responses. Consequently, a request is made by a system and it usually chooses to do something else for awhile rather than wait for the response. Such behavior means that an unsolicited input from a network can result in the creation, deletion or other modification to a shared resource. For example, consider what occurs with a *listen()/accept()* sequence. From the UNIX Programmer's Manual pages [UNIX 1983]:

listen (2) - listen for connections on a socket
accept (2) - accept a connection on a socket

These UNIX IPC services are used to establish an IPC server. The server uses a *listen()* system call to inform the system that it intends to receive connection requests from one or more clients. It uses a standard socket descriptor which it created to do this, however, the *accept()* system call returns a different socket descriptor. The listen socket is essentially used as a template for connection requests to describe the communications for subsequent *server/client* conversations. Each new request for connection from a potential client results in the creation of a new socket structure. The new socket is created asynchronously and initiated by a client process that can reside on a different host. When the connect request is received by the network, a new socket is created using the *sonewconn()* kernel function and queued to the server process using the listen socket. The new socket is subject to the same mutex considerations as sockets created via user requests and can also be aborted before they are completely established for protocol engine reasons. This is particularly interesting in a multiprocessor system since all of this occurs concurrently along with other processing as well.

Another example of remote initiation of resource modification is dynamic reconfiguration of network topology. Internetworking situations can result in dynamic modification of routing information or other connection state initiated by unsolicited network input. As an example, if a host or network suddenly becomes unreachable due to a system failure, Internet systems communicate with each other and direct that existing connections through failed systems be aborted. This too causes interesting dynamics in a multiprocessor system.

5.4 Concurrency Issues With Multiple Users and User Processes

The UNIX operating system supports the notion of many simultaneous users and each user can *fork()* a number of user processes executing on the user's behalf. The UNIX model permits, in fact encourages, processes to *fork()* children and allow them to share system resources that they initiated, for example standard input, standard out, and newly created server sockets. Typically, a parent process allocates the resource, then forks the child, then no longer accesses the resource. Typically, child processes do not share these resources although the possibility is available to them. However, typical execution cannot be the model, and since these resources are sharable, it is assumed that they will be shared, intentionally or otherwise. Therefore, once a socket is created, it can be accessed by one or more processors concurrently in the form of user processes and their child processes requests.

The monoprocessor kernel implements some consideration for user processes sharing a socket resource using busy and want flags on socket buffers and *sleep()/wakeup()* semantics. If multiple processes attempt to read from a socket, they are serviced on a whoever gets there first basis. Note this is not First Come First Serve, for if a socket is marked busy when a user process makes a request, the user process is put to sleep and awaken when the socket buffer is available. If multiple processors attempt this, they are awakened in pseudo-random order. Note, this is an area where *real concurrency* causes different, but semantically compatible behavior on a multiprocessor. *sleep()/wakeup()* issues are further discussed below.

5.5 Connect Semaphore

Often, protocol connection establishment is an asynchronous activity in protocol management. This is because, for example, protocol handshakes must be completed before the connection is completed. This is also true for disconnection and often a protocol engine requires that state information remain available until the disconnection is complete. In a monoprocessor kernel this is accomplished via *sleep()/wakeup()* semantics. In DYNIX this is accomplished by using semaphore structures. A connection semaphore is included in the socket structure and when a connect is requested, the protocol engine is called with a PRU_CONNECT request and the process is queued on the connect semaphore.

Note it is an easy mistake to assume that the above semantics describe simple replacement of *sleep()* with *p_sema()* and *wakeup()* with *v_sema()*, but this is not the case. Care must be taken in porting *sleep()/wakeup()* semantics to a real multiprocessor. In a monoprocessor, the awakened process does not run until the kernel relinquishes the processor. In a multiprocessor system, a *v_sema()* can make a process run immediately. In fact, the awakened process can start even before the return from the *v_sema()* call completes.

6. Things That Can Go Wrong

The purpose of this section is to give the reader an appreciation of the things that can go wrong when developing IPC in a multiprocessor system. The tasks is complicated by the objective to take advantage of as much existing code as possible, and absolutely maintain interfaces and user services. This is the objective of the DYNIX port, and it fundamentally is true to the UNIX 4.2bsd IPC implementation.

DYNIX adheres to the concurrency model described above as much as possible however, the model does not fall out of the port easily and great care is required to avoid multiprocessor pitfalls. Even so, it isn't proven until it actually works in a true multiprocessor system. Additionally, it must also be proven in a true networking environment since network processing happens as a result of other machines on a network. For this reason considerable stress testing is required to ensure completeness. The stress tests used must exercise heavily all of the error paths in a concurrent system. For the most part, this was done and needless to say, some errors were uncovered. The following discusses the types of errors that were encountered, how they were analyzed, and finally how they were fixed.

6.1 Mutual Exclusion

Mutual exclusion is never having to say %@#$^!!&^. It is the fundamental requirement to consider in multiprocessor systems. In fact, it is required in any system that allows multiple execution streams to modify the same data structures. In such systems, portions of the execution streams are sensitive to concurrent modification of shared resources. These form **critical sections** of execution during which concurrent modification must be avoided by mutually excluding other processes from conflicting critical sections of execution. Critical sections typically occur whenever shared data is modified. Some critical sections, particularly in a porting effort, are difficult to notice. For example, it is common practice to initialize local variables with their declaration (e.g. int foo = bar;). If the local variable is initialized using a shared data structure, the initialization must be moved to occur *after* the lock is acquired.

When mutex is required, it is critical for predictable system operation. Inadequate mutex can cause **race conditions** in which different execution streams "race" for a resource and if one looses, data is compromised. Because of race conditions, some errors are difficult to find because a system becomes nondeterministic. An error may not occur predictably after any particular sequence of execution although typically some "impossible" condition eventually occurs and the system fails. Most problems encountered in development of a multiprocessor system are due somehow to improper mutual exclusion.

6.2 Why the Monoprocessor Mutex Model Doesn't Work

Mutex is required even in monoprocessor environments if multiple streams of execution are supported. This is a common case, for example, in systems that allow interrupts and device handlers that modify shared memory areas. The UNIX monoprocessor model of mutual exclusion uses processor priority level (SPL) to mutually exclude one stream of execution from another. Streams of execution are blocked from critical regions by "raising" the SPL of the hardware. This prevents a process that executes at a lower or equal SPL from modifying a data structure until the SPL is lowered. For example, in UNIX4.2bsd code that requires mutex of network processing raises the SPL to SPLNET to prevent network code from asynchronously accessing shared structures as a result of unsolicited network input (or timer event). This is an example of a simple case of mutual exclusion requirements since one and only one processor still does just one and only one thing at a time. Obviously, actual concurrency is not generally achieved with a monoprocessor model.

A common pitfall in assessing the methods to use in porting a UNIX kernel to a multiprocessor environment is to assume that the use of SPL identifies all of the critical sections and simple replacement of SPL management with appropriate mutex calls is sufficient. It is not. DYNIX provides SPL-like capabilities on a per processor basis via the mutex calls. However, the SPL mechanism does not provide mutual exclusion of other processors executing at any other SPL. The SPL mechanism does provide simple mutex in a monoprocessor system, but it is by no means equivalent to a lock or gate type mechanism. The semantics of SPL management are quite different than the semantics of locks and gates. Simply replacing SPL calls with what seems to be appropriate locking and unlocking calls quickly causes dangling references, deadlocks, or other multiprocessor problems as discussed below.

6.3 Dangling References

A common error is that of **dangling reference**. This refers to the situation where a structure is referenced after the resources it uses are released. When multiple processes share a structure, the structure cannot be released until all processes are finished with it. Consider that after the resources are returned to the system, they can be used for something entirely different by another processor. The fundamental shared data structures in the IPC kernel are *mbuf's*. They are used for every dynamic storage need in the IPC kernel and managed separately from other kernel memory space. A released *mbuf* is placed on the front of the available *mbuf* free list and so typically is reallocated to another processor immediately in a multiprocessor system.

One of the dangers encountered in the port of a monoprocessor UNIX4.2bsd is that there exist dangling references that are harmless in a monoprocessor system. Various structures are released and later referenced within the same system call or network input process. Kernel structures are sometimes released and later used to reference, for example, a sleeping process that is waiting for a state change on the resource. The correctness of such coding practice is arguable even in a monoprocessor system, however it does not necessarily cause system failure. This is because the system call is typically executing at a raised SPL. The released *mbuf* is not reallocated in a monoprocessor until the system call lowers the SPL. Therefore, the referenced information is still useful even after the resource is released. This is not the case of course in a multiprocessor system since resources become available to other processors immediately upon being released.

Dangling reference can also be introduced with general mutex modifications. A simple example illustrates such a dangling reference. Consider the following generalized program structure:

```
LOCK resource;

switch(cmd) {
CASE 1:
        { }
CASE 2:
        { }
CASE N:
        { }
}

UNLOCK resource;
```

This appears straight forward enough and reasonably structured. However, if there exists one or more paths through the switch cases that releases the resource upon which the LOCK operation acts, the UNLOCK operation uses a dangling reference to a nonexistent resource. Since that resource is immediately available to other processes executing on other processors, the results are unpredictable. The UNLOCK operation above makes its state change to an *mbuf* that is being used by another processor for something else. Algorithms must accommodate cases where the referenced resource *disappears* during the critical section and therefore the lock which seems necessary to unlock must in fact not be unlocked due to dangling reference.

A side note in these cases is that the SPL is separately managed here. Mismanagement of an SPL can result in a processor becoming stuck at a particular, high SPL. In a multiprocessor system this might go unnoticed since there are other processors to take over the lower SPL tasks, but in a monoprocessor system being stuck at a high SPL can cause a form of deadlock discussed below. In DYNIX, SPL is usually managed via the *p_lock(), v_lock()* semantics, but in these cases where *v_lock()* cannot be called, *splx()* is used explicitly to restore SPL.

6.4 Deadlocks

Deadlocks refer to the phenomena where one or more processors are put into a position such that they can not make progress due to faulty mutual exclusion. Several types of deadlocks exist including sequential deadlocks, asynchronous deadlocks, and deadlocks involving multiple resources.

Simple **sequential deadlocks** often show themselves even in monoprocessor situations. They occur when a lock is acquired and later in the same execution path, without releasing the lock, another attempt is made to acquire the lock. In porting code this can happen due to use of common modular routines and macros. For example a resource close routine can be called from several places in the code some of which already possess appropriate locks and some that do not. A solution to this problem is duplication of the routine with appropriate locking semantics. This is done in a few places in DYNIX, but generally avoided if possible.

An **asynchronous deadlock** occurs some time after the fact, on a subsequent reference to the shared resource. This happens for instance if an error path out of a system call or network process does not properly release a lock or gate. In such a case, the lock is held never to be released and the next attempt to acquire the lock deadlocks. Note, that it is not usually the same sequence of code that left the lock held that ends up spinning on it. This makes debugging difficult without appropriate tools. One useful tool for this case is a circular buffer to record lock access activity. Such circular buffers were used in the development of DYNIX.

Deadlocks are more subtle with semaphores since an affected process essentially becomes inactive, waiting for a semaphore which is never incremented. Depending on the number of processes that share the semaphore, eventually all processes that access that semaphore end up waiting.

Another useful tool for deadlock detection is **processor lights**. The Balance 8000 displays LEDs on the processor boards. These lights are operated by the kernel scheduler. When the processor becomes idle, its light is turned off. When it becomes active, it is turned on. Experienced developers can determine a lot about the operation of a multiprocessor system by watching the lights. If a processor starts to spin on a lock, its light stays on brightly. If multiple processors deadlock, multiple lights stay on brightly. In systems of numerous processors, deadlock may not be obvious from the console or terminal keyboard, since it may only deadlock two of the available processors. In such cases, console and terminal response are unaffected and processors that are not deadlocked proceed. CPU intensive processes also keep the lights bright but typically occasionally flicker and often migrate from processor to processor. If all processes in, for example, a test suite become inactive due to improper semaphore management, all of the lights stay out. The method of using lights is often subjective, but proved useful in characterizing multiprocessor system behavior.

Deadlocks that involve multiple resources are more difficult to detect, and sometimes do not occur in a monoprocessor system since the critical sections are not really executed concurrently. The classic case of **multiple resource deadlock** involves two processes that require two shared resources, say locks. One process has resource A and needs resource B. Another process has resource B and requires resource A. This situation easily deadlocks unless care is taken to avoid it. One approach to avoid deadlocks like these is to always acquire resources in the same order. For example, if the above locks are always acquired A then B, then deadlock does not occur. Whichever process gets to A first, blocks any other process that requires both resources from getting resource B. In some cases it is possible to open a **window** during which another process can acquire both resources in the prescribed order. Allowing such windows is sometimes convenient but must be done with care since it introduces race conditions. Typically, a state must be reliably checked after the resource is acquired to ensure that its state did not change during the window. The process that allows the window indicates that it still has a reference to the structure so the winner of the race condition does not pull the resource out from under it. Consider, the state of the resource cannot be checked if the resource is deallocated. Windows also have performance considerations since they usually introduce additional mutex requirements.

There are also subtle deadlocks that do not occur in multiprocessor systems, but can occur in monoprocessor systems. For example, consider a process at a low priority level acquiring a resource that is eventually demanded by a process at a higher priority level. In the multiprocessor case, generally the lower priority process is eventually allocated processor bandwidth and allowed to complete execution of the critical section. However, the same situation executed in a monoprocessor system deadlocks due to the fact that the high priority process does not relinquish the processor to the lower priority process to complete the critical section. For this reason, testing of multiprocessor implementations must necessarily include testing for the case of N = 1.

6.5 Nested SPL's

In a monoprocessor UNIX system, SPL's can be "nested". This means that an execution path modifies an SPL, then later modifies it further. This is done in the kernel for instance due to use of modular routines and general purpose macros. Consequently it is possible to set an SPL to a particular level, and later execute a common subroutine which executes a similar SPL request. The affect in a monoprocessor system of the second SPL call is essentially NULL.

If the same resource is being accessed and the *spln()* calls are simply replaced with *p_lock()* calls, immediate deadlock occurs. It is possible to nest locks by referencing multiple locks but this encounters the multiple resource mutex management issues described above. Also note that in a monoprocessor system where nested SPL's are used for mutex, it is not always necessary to execute an *splx()* for every nesting level. Often it is sufficient to execute one *splx()* in order to return

to the user priority level. This is obviously not the case for nested locks since such an algorithm leaves locks held. This results in an asynchronous deadlock the next time the lock is referenced.

Nested SPL's do not always raise the SPL. It is possible to nest *splx()* calls which restore the SPL. In a monoprocessor system multiple calls to *splx()* to restore the same SPL are harmless, but if mapped one to one to UNLOCK operations in a multiprocessor system mutual exclusion is easily compromised. Observe that what occurs in the above situation is an extraneous unlock of a resource. This can be difficult to find if nested deep in the code because the damage does not occur while executing the erroneous execution stream. The erroneous execution stream essentially unlocks another processors lock and thereby opens up another processor's critical section.

6.6 IPC Calls to IPC Services

The IPC kernel also calls IPC services which introduces another source of potential deadlock. For example, a powerful notion in a networking system is the ability to access local services using the same addressing mechanisms as remote services. This is accomplished via a loopback path. Loopback allows local IPC identical to remote IPC. This means that the output side of the communications subsystem makes use of the services of the input side. This is done using the loopback driver which performs both as a sender and receiver. Loopback operates by taking packets from the transmit side of a network interface and placing them into the receive side of another network interface. From the protocol engine viewpoint, there is no difference to the data flow.

The 4.2bsd implementation also implements an Address Recognition Protocol (ARP) within the Ethernet driver which presents an interesting case. The network interface receiver interrupt accepts a data packet, determines that it is an ARP request, and calls an appropriate routine. The ARP data base is searched and it is determined whether or not a reply is required. If a reply is required a response is queued to the appropriate network interface transmitter which executes concurrently. The Ethernet output routine also queries the ARP data base to determine if address resolution is required. In effect, the Ethernet receiver initiates an Ethernet transmission. In the monoprocessor model, this by necessity is accomplished by a single processor at the same SPL and thus mutual exclusion is not an issue. In DYNIX, the transmitter and receiver execute concurrently and share the ARP data base.

6.7 Deadlock Avoidance Example - Pipes and Socket Peers

An example of how deadlocks are avoided in DYNIX is the implementation of pipes. Pipes are a particularly powerful, very popular UNIX capability. In the UNIX 4.2bsd kernel they are essentially a limited special case of the standard IPC mechanisms. Limited because they are unidirectional. In a monoprocessor implementation of UNIX 4.2bsd, multiple processes are involved with pipes, but not multiple processors. Therefore, the kernel can complete the data transfer from one user process, into kernel buffers *(mbufs)*, and queue them for the receiving user process before the receiving user process starts receiving the data into its user buffers. However, in DYNIX both processes execute concurrently.

The *pipe()* system call creates two socket type file descriptors. One socket is maintained to keep track of the send side of the pipe and another is maintained to keep track of the receive side. Consider the straight forward approach. Process A is a producer of data to put into the pipe and process B is the consumer. Process A executes a *write()*. The AF_UNIX domain protocol engine locks the sender socket and finds the receive side. It then locks the receive side and places the data into it, unlocks it and notifies any processes waiting that data is available. A consumer process does a *read()*. A concurrently executing AF_UNIX domain protocol engine locks the receive side of the pipe in order to check the state and take data out of the queue. It then must update the send socket to notify it that the data is accepted in order to affect flow control. Notice that both processes acquire both resource locks in a different order. This results in deadlock the first time a process attempts to send data while another processor is receiving data from the same pipe. It is not necessarily appropriate to open windows in this case since the order in which the states are modified is important and potentially multiple windows are required on

every send or receive request.

DYNIX pipes avoid deadlock by sharing a mutex structure in something referred to as a **socket peer**. A socket peer is established at pipe creation time and contains the lock structure used to mutex both sockets. If simultaneous read and write requests are made, one blocks out the other while it quickly modifies the socket state. Note, data is copied between kernel space and user space without a lock being held. This is particularly significant when one considers that the user process' space can be swapped out in a virtual memory system such as that supported by DYNIX. It also means that typically the socket is locked only while the send and receive queue parameters are updated, not while data is being moved.

6.8 Deadlock Avoidance - Network Processing and Socket Peers

Socket structures are also accessed from the network concurrently. A protocol engine can be emptying the send queue while a user process is attempting to place a request into it. Protocol engines use the send queue to define flow control, and keep data in the queue until its delivery is assured. The queues can be modified by timer events or as a result of unsolicited network control messages. The protocol engine also references a protocol control block that contains addressing and state information. The protocol control block is used by the protocol engine to find the associated socket structure. It is cumbersome to simply provide a lock on the protocol control block and a lock on the socket because access to these structures is both network to kernel and kernel to network. This means that sockets are accessed both through protocol control block reference and through socket references. Notice that this presents multiple resource deadlock concerns similar to those discussed above.

The approach taken in DYNIX is again to define a **socket peer** mutex structure similar to those used for pipes only now the sharing entities are socket management and network protocol engines. The socket peer is created at protocol "attach" time and is used to mutex both the socket and the protocol control block. Therefore when critical sections are necessary in the protocol engine related to a particular socket structure, user processes that access that socket are locked out. Typically this is for a short time until data gets queued or dequeued from the interface. Note, that unsolicited data also implies unsolicited control data, and the shared structures can be modified, even deleted due to a network input. Therefore, once a protocol control block is found by the protocol engine, its socket peer mutually excludes the user from changing its state until its action is completed. Other user process socket structures can access other sockets and other network functions while a protocol engine works on a particular socket. Also other protocol engines can be executing.

To manage creation and deletion of socket peers, a reference count is maintained per socket peer. The socket peer can be referenced by the user process space, by the network binding space, by the network input space and by special events such as timeouts and unnatural termination requests. An example of an unnatural termination requests is the closure of a route to a foreign hosts. Recall that in the Internet this results in peer level IP layers exchanging Internet Control Messages (ICMP) which results in all currently active connections to a suddenly unreachable destination to be aborted.

To appreciate the issues in concurrent processing consider the worst case in this model. The user process, in fact several user processes, can be attempting to close the resource, due to termination for example. The route could have become unreachable such that ICMP messages have directed its abortion. A timer event could expire indicating a complete disconnection which results in the deletion of the connection. All of these events can occur simultaneously and be assigned to different processors to execute concurrently. Resources must remain available until the last reference is made to them to avoid dangling reference. Obviously, a monoprocessor implementation does not have to consider all of these events occurring simultaneously since it can essentially serialize the requests and deny or ignore those that it schedules after deletion.

The port to DYNIX of the IPC is very careful to arrange deletion algorithms such that references to a resource are not made after the resource is removed. This means for example that when a socket is deleted, the reference count of the socket peer is checked to ensure that there are no longer any references to it before it is released. If there are references, then it is unlocked in order to allow the last references to be removed. This occurs for example if the socket is being closed while the network is processing an input packet (control or data) that is bound for it. The algorithms discover later that the socket has been closed since the protocol engine that executes on behalf of the user to close the socket, typically changes the state of the protocol control block to indicate disconnection. The process that makes the last reference to the socket peer releases the resources. Note the mutex considerations here. As described above, it is not proper to UNLOCK a lock structure that has been deallocated. The general socket-peer deletion algorithm becomes:

```
LOCK socket peer;
release appropriate resources;
increment socket peer reference count;
do appropriate protocol action; /* which may cause deletion */
decrement socket peer reference count;
if(last reference)
        release socket peer resources;
        restore appropriate SPL;
else
        UNLOCK socket peer resources;
```

Also as in the case of the generalized system call example where the lock structure is released, returning to the appropriate SPL is considered. Since the DYNIX *v_lock()* interface returns to a specified SPL, those semantics are emulated in cases where *v_lock()* is not used.

6.9 Concurrency and Shared Lists

A common method of sharing information is in the form of lists. These can be managed as queues, or static or dynamic data structures. Examples of lists maintained in the kernel include lists of data streams, pending connection requests, and protocol control blocks used for polling, address binding and synchronization of system resources.

A example of this is the list of protocol control blocks maintained for active TCP ports. These ports can be in various states such as newly created, not yet connected, listening for connection requests, connection established, partially disconnected and finally disconnected and on its way out. The list is headed by a TCP Control Block structure and consists of a linked list of TCP Control Blocks that contain addressing and state information. When a TCP packet arrives, it is first demultiplexed by the network interface (viz. Ethernet) driver and placed into the Internet input queue. The Internet input queue handler, *ipintr()*, executes concurrently at a lower SPL. The Internet protocol engine quickly recognizes a TCP packet and calls the *tcp_input()* routine. The *tcp_input()* routine scans the TCP control block list to determine if it knows about the destination specified in the packet (i.e. the port number). The protocol control block list is concurrently accessed by user processes that add and remove references and also by the timer event routines which periodically scan the list to change the protocol control block state (e.g. decrement a time counter, execute a protocol time out event,..). Also consider the *sonewconn()* case which results in the input side of a protocol engine adding a new protocol control block to the list. Obviously, both the protocol control block list and the particular protocol control block require mutex.

A socket peer approach is not appropriate in this case since there are many peers in the list. Therefore the straightforward approach is for *tcp_input()* to lock the TCP control block list, find the appropriate protocol control block and lock it. However, note what happens if the

protocol control block is associated with a socket that is in the process of being deleted by a user process request. The kernel on behalf of the user process locks the socket and the protocol control block (using their socket peer), and then locks the list in order to detach the protocol control block from the list. This creates another deadlock.

DYNIX avoids deadlock here by assuring that in those cases that require both the list and elements in the list to be locked, the locks are always acquired in the same order. First the list is locked and then the element. This order is chosen because locking the element first and then the list results in many more windows being opened by processes that traverse the list. Opening a window allows a race towards acquiring the lock for the element. Care is required that this race can be lost and state information not be compromised. When an element is being deleted, its state is completely updated before a window is opened in order for it to be detached from the list. This avoids access by other user process requests. To avoid access from the list processing functions, further indication is used. Each entry contains a pointer to the head of the list. A NULL value in this pointer indicates that the entry is on its way out. If a processor is traversing the list and wins a race for the lock of a control block that is on its way out, the NULL head pointer value indicates that there is no processing appropriate for this entry. Care is taken not to reference protocol control block state information after it is possible to delete it and information such as the pointer to the next element in the list is maintained across the call. The pointer to the next element is acquired with the lock held to avoid a dangling reference when the control block is deleted. The control block is then unlocked, thereby allowing the deletion to continue.

Some actions of list processing result in deletion of the element under consideration. This means for example that list processing calls list deletion routines. Alternate detachment routines are implemented which detach elements with the list locked. If convenient, these routines are called, otherwise, windows are permitted and algorithms ensure that no dangling reference occurs. The list processing increments a reference count on the lock structure (e.g. the socket peer) in order to avoid it being deleted along with the control block.

6.10 Concurrency and Time Outs

Time waits for no LAN! Many functions of practical protocol engines depend upon the system (i.e. the kernel) to keep track of time and its relation to state information. It accomplishes this by issuing a time out event at regular intervals. Specifically, transmissions time out and are assumed lost, retransmissions are submitted, round trip delays are calculated and connections are abandoned due to certain time limits being exceeded. In a monoprocessor environment everything literally stops during a timeout and the entire state of the protocol engine examined and acted upon. This is not the case in a real multiprocessor environment and considerable attention is required for protocol timeouts.

At each time event, the domain structure is referenced to find the appropriate *protosw*[] array and the time event routine for each protocol. There are two such events, the fasttimo event every 200 ms. and the slowtimo event every 500ms. Such events are used to calculate protocol timeouts, etc. Note that not all protocols maintain states based on time events. For example, non-guaranteed data gram delivery offered by the Internet UDP protocol specifies NULL time out event routines in the *protosw*[] entry.

The TCP time out routines are also concerned with list processing similar to the *tcp_input()* routine. Time out events can result in deletion of protocol control blocks and so the reference counting algorithm is used. The TCP timeout routines additionally avoid spinning on protocol control block locks. It does this by using the conditional *cp_lock()* interface. The assumption is that if a protocol control block is locked, it is undergoing state change that most likely includes timer state changes. This is true in most instances, and it is noncritical if a timer event is missed.

6.11 Cached Temporary Data

The term **cached temporary data** refers to a situation where shared structures are used to temporarily store data during a system call that has network interrupts blocked out on a monoprocessor system. The shared structure is restored to the appropriate value before network processing is continued. A specific example of this is the use of globally referenced structures used to hold temporary values. These structures are moved to the local parameter scope (i.e. the stack) to avoid sharing them when not necessary. A more subtle example of this is the use of UDP protocol control blocks to store the address of an outgoing datagram. This information is passed to the UDP transmit routines and used to create the UDP packet header. After the UDP datagram is created and queued for output, the protocol control block is restored. This does not work correctly in a multiprocessor system because concurrent references are made to the protocol control block by the network input routines searching for address matches. For the time that the protocol control block is being used for temporary storage of temporary binding information, addresses do not match. In the case of UDP data grams, this means that they are not delivered. This situation is easily tested by having multiple UDP clients sending datagrams to a single UDP echo server. This typically results in simultaneous send and receives and consequently datagrams are frequently missed.

6.12 Data That Corrupts

Improper mutual exclusion can cause situations where network interfaces put data where it shouldn't go. This is particularly true when asynchronous DMA is supported and exacerbated by shared memory in multiprocessor systems. Corrupting data buffers can result in application programs failing, but it can also result in protocol failure since at some level, protocol control information (e.g. headers) is generic data. Furthermore, since *mbuf*'s are also used for many IPC kernel structures they too can be corrupted by faulty data placement which can cause unpredictable behavior. There are safeguards against errant kernel code modifying code space, but these safeguards do not usually apply to system i/o such as DMA from network interfaces.

In a multiprocessor system that supports DMA, it is sometimes necessary to narrow down such a problem to the network subsystem. It is helpful that the kernel typically uses only IPC kernel buffers, *mbufs*, for IPC kernel managed data. It is also helpful that all network buffers start on an *mbuf* byte boundary (modulus 128) and all DMA requests are *mbuf* size (128 bytes) or less. These observations can be applied to verifying that a network interface is compromising system memory.

Another approach to debugging errant data transfer by a network interface is to use tests that transfer identifiable patterns. This approach helps in general to debug data movement implementations but there are special considerations particularly applicable to network system development. Patterns should be chosen with care. They should be easy to recognize in an environment where lots of data is present. In network system development, it is often necessary to view *exactly* what goes on to the wire. Therefore patterns should be readily recognizable in a primitive form of representation (viz. hexidecimal). For example, it is useful if the patterns "spell" recognizable "words" using hexidecimal digits. Networking development is often concerned with bit and byte ordering since it deals with heterogeneous machine architectures [Kirrmann 1983] and network canonical forms. This implies for instance that test patterns should not be palindromes and be able to indicate byte swapping errors. There are several interesting hexidecimal patterns which meet these requirements.[9] A particular pattern chosen in DYNIX development is hexidecimal FEED FACE DEAD BEEF. Using this pattern in various portions of network data packets assisted the tracking of data flow from machine, to network, to machine. If byte swapping errors occur, FEED FACE becomes EDFE CEFA or CEFA EDFE (depending on whether the error is on a short or long). If network data is incorrectly placed in the destination and something breaks, FEED FACE DEAD BEEF is evidence enough to suspect problems in the flow of

9 Try egrep '^[a-f][a-f][a-f][a-f]$' /usr/dict/words.

network data.

7. Conclusions

DYNIX is not *the* solution to loosely coupled functions on a tightly coupled engine, but it certainly is *a* solution. It works and works well. The problems encountered in porting the described UNIX 4.2bsd IPC model to the described DYNIX multiprocessor model are similar to problems encountered in any multiprocessor system. However, as this paper describes, the nature of networking introduces considerations such as unsolicited data and remote initiation of shared resource modification that exacerbate common multiprocessor problems and manifests them in ways particularly applicable to networking. The lessons learned and the solutions implemented described in this paper are considered useful for further research and development in general multiprocessor technology.